A Fast Method for Traveling Salesman Problems

PAPERS_0347
[Author]
[Author Affiliation]

Abstract

We present a dynamic method for geometric instances of the Traveling Salesman Problem with Neighborhoods (TSPN). To improve scalability, we implement it using a new self-adjusting binary search tree. Through testing, we demonstrate fast execution and better quality than the random insertion heuristic. Our method offers an uncommon combination of simplicity, performance, scalability and flexibility; making it appropriate for use in many real-world applications.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques —Graphics data structures and data types; E.1 [Data]: Data Structures—networks and graphs.

Keywords: Traveling Salesman Problem, self-adjusting data structures, binary trees

1 Introduction

In this paper, we present a dynamic method for geometric instances of the Traveling Salesman Problem with Neighborhoods. The Traveling Salesman Problem (TSP) is a classic problem in computer science. The problem is this: given a set of cities, find the shortest route which passes through each city exactly once, and returns to its starting position. A path between two cities is called an arc, and the complete path is called the tour. Figure 1 gives a visualization of a TSP tour. The Traveling Salesman Problem with Neighborhoods (TSPN) is a variation where each city is represented by a region rather than a point position. The Dynamic Traveling Salesman Problem (DTSP) allows the distance-cost matrix to vary arbitrarily over time, and cities may be added and removed.

The family of problems is NP-hard, so finding the optimal tour is impractical. As a result, the problem has provided a good platform for developing optimization techniques over the years. Recently, DTSP methods have seen active development because of their relevance to dynamic navigation. In particular, dynamic vehicle routing adds payload limits, as well as pick-up and drop-off constraints, to the DTSP problem definition.

1.5” margin for ACM Copyright

We present a fast new method for producing tours for the TSPN, applicable when neighborhoods can be suitably approximated with axis-aligned bounding boxes. The method is dynamic in the sense that neighborhoods can be added or removed at will; however, it applies to geometric problem instances because the distance-cost is modeled as a function bound by the triangle inequality rule. We apply a new self-adjusting binary search tree to implement the underlying data structure. Our heuristic is reasonably easy to implement, but still offers a rich set of features relative to typical TSP heuristics. Experimental results are presented.

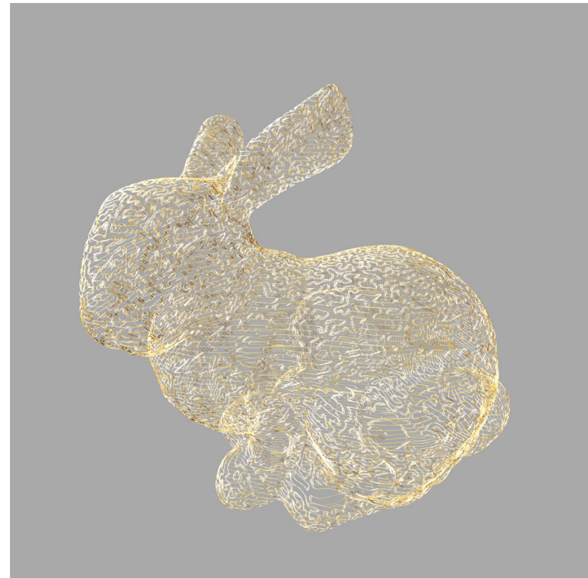


Figure 1: The Stanford Bunny: A Traveling Salesman tour of the model’s vertices rendered as a golden wire.

2 Shuffle Trees

In this section, we introduce a new self-adjusting binary search tree, which we call a shuffle tree. The name comes from how it converges to a weight-balanced state. During each access, a rotation may be probabilistically applied. Just as cutting a deck a large number of times results in a randomized ordering of playing cards, a large number of shuffle tree accesses results in a good node configuration.

Scapegoat trees [Galperin and Rivest 1993] and splay trees [Sleator and Tarjan 1985] are related binary trees that, like shuffle trees, require neither parent pointers nor balancing information to be stored at each node.

Splay trees are the most popular self-adjusting binary search tree. During each access, the node subject to the access is moved to the root by a specialized series of rotations known as splaying. Frequently used items collect at the root of the tree, allowing for fast access to them.

The scapegoat tree monitors each access to detect a long path which exposes poor balance in the tree. A node is then identified along the path under which balance is particularly bad, and the sub-tree under the node is rebalanced.

Like scapegoat trees, shuffle trees sample tree depth to identify regions of imbalance. At the beginning of each traversal, we set an integer count-down value, *I*, to a random number in the range [0, N-1], where N is the number of nodes in the tree. As we iterate from a parent node to its child, we decrease the counter with $I := \frac{1}{2}(I-1)$. When the counter equals zero, the current node becomes a candidate rotation pivot. If we need to iterate past the candidate pivot, then we will commit to a rotation. We rotate the pivot away from the direction of traversal. Counters are reset for each traversal. If the counter does not reach zero, then no rotation occurs. Shuffle trees use the rotation operation depicted in figure 2.

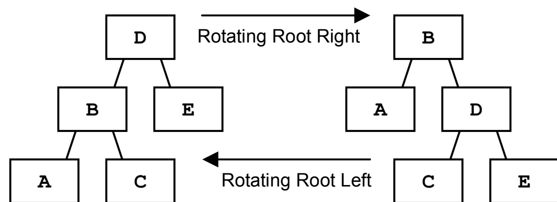


Figure 2: A rotation at the root moves some nodes from one sub-tree to the other, to affect balance in the tree.

As search depth increases, the likelihood of a rotation increases. No rotations will occur beyond depth lgN and the counter requires lgN random bits per operation. Shuffle trees also record their size, so that the counter can be set. No balancing information needs to be recorded in tree nodes.

Algorithms for search, insertion and deletion from unbalanced binary search trees can be easily modified to support this balancing operation. Insertion and deletion require traversals that stretch from root to leaf, so they provide good opportunities to conduct balancing activity. Searches may not navigate to a leaf; if nodes of interest are clustered near the root, then deep searches will not be required and rotations can occur less frequently. Like splay trees, the balancing method for shuffle trees can be encoded into a simple traversal function [Sleator 1992]. All tree operations defer to this traversal function, making implementations very simple. A reference implementation of associative data structures based on shuffle trees is available at the author's website at http://website/website.com.

A thorough study of shuffle tree properties is beyond the scope of this paper; however, we will argue informally that the shuffle tree is resistant to attacks and maintains *O*(lgN) access performance.

Imagine that an adversary wishes to ruin the tree by forcing rotations to be applied that would linearize it. If a binary tree becomes linearized, then all operations become *O*(N), otherwise all operations execute in *O*(lgN) time. Let us consider the shuffle tree to be a kind of weight-balanced tree, where the probability of a node being accessed is its weight. The adversary would need to access nodes in the tree in a specific manner such that nodes are consistently moved by rotations from sub-trees of lesser weight to sub-trees of greater weight. To simplify the discussion, assume that each node is equally imbalanced; that the probability of traversing into the heavier sub-tree is always P_W, where $P_W > 0.5$.

The probability that an adversary can select a node to force a rotation that impairs balance is

$$(1-P_w)\prod_{i \in A}(1-p_i)$$

where A is the set containing the rotation pivot and all its ancestors; and p_i is the weight of a node *i*, the probability of it being accessed. Since $P_W > 0.5$, the adversary never has a better than fair chance of hurting balance. The chances might seem to be better when P_W is close to 0.5, meaning that the tree is initially well-balanced. However it is a property of weight-balanced trees that shallow nodes tend to have larger weight, so the probability of forcing a bad rotation diminishes fast with depth of the rotation pivot. Of course, the probability of a shallow rotation is significantly less than the probability of a deep rotation because of the way the counter operates. At the other extreme, in a poorly balanced tree, the probability of forcing a bad rotation is low because P_W is large. Over a large number of accesses, the probabilities cascade to defend against adversaries and to approximate weight-balance.

Because the shuffle tree applies rotations sparingly, it is well-suited for applications where a rotation is an expensive operation. This includes our application described in subsequent sections.

3 Preliminary Material

Because of its rich history, the literature pertaining to TSP heuristics is vast. A complete summary of existing work is beyond the scope of this paper; for that purpose, good surveys exist [Applegate 1999; Johnson 2002]. We will only discuss the methods most relevant to our technique.

The methods described by Rosenkrantz [1977] form a family known as "insertion heuristics." The variations differ in the order in which cities are inserted into the tour, but the insertion algorithm is consistent. To insert a new city into an existing tour, one arc needs to be removed and two added. A search is conducted to find the specific arc for replacement such that the increase in the tour length will be minimized.

Another popular and successful family of heuristics is known as *k*-opt refinement. These methods usually first build an initial tour by using some fast means. To improve quality, the tour is refined using an operation known as a *k*-opt flip. In this operation, *k* arcs are removed from the tour and replaced with *k* new arcs in an alternative configuration, resulting in a shorter tour. Because a tour is directed, a *k*-opt flip may require that a section of the tour is reversed in the process. The famous technique of Lin and Kernighan [1973] is a variation where the *k* is allowed to vary. The 2-opt methods tend to be popular because they are simple to code and perform well in practice. Figure 3 provides a visualization of a 2-opt flip operation. Under favorable conditions, a 2-opt solution can produce tour lengths within a constant factor of the optimal [Chandra et. al. 1994], but recent

theoretical work finds that a complete 2-opt solution can take an exponential number of iterations [Englert et. al. 2007].

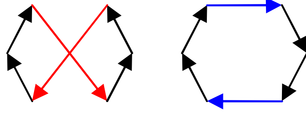


Figure 3: A 2-Opt flip operation: The two red arcs in the tour on the left are replaced with the two blue arcs as shown, reducing tour length. A sub-section of the tour reverses its direction.

Both, insertion methods and *k*-opt methods, require distance functions that obey the triangle inequality. Simply stated, there can never be a shorter path between two cities than the single arc between them.

We can easily generalize both these families of TSP methods for the TSPN, even if the neighborhoods overlap, when it is reasonable to approximate neighborhoods with their axis-aligned bounding boxes. The furthest distance between two bounding boxes can be used as the distance function. We calculate this distance using the methods of Patel [2007], which we review here.

Regarding notation, we write a vector as $\underline{V}$; its component in dimension i is given with v_i. An axis-aligned bounding box, R_0, has a minimum coordinate of r_0^{i-} and a maximum coordinate of r_0^{i+} in dimension i.

There exist two vectors, $\underline{M}$ and $\underline{N}$, whose components are calculated as a function of two axis-aligned bounding boxes, R_0 and R_1:

$m_i = \max(r_0^{i+} - r_1^{i-}, r_1^{i+} - r_0^{i-})$

$n_i = \min(r_0^{i+} - r_1^{i-}, r_1^{i+} - r_0^{i-})$

The furthest Manhattan distance between two bounding boxes is:

$$D_F(R_0, R_1) = \sum_i m_i$$

As neighborhoods converge to points, the furthest Manhattan distance converges to the regular Manhattan distance between those points.

We will also use the nearest Manhattan distance to aggressively cull our spatial searches. We calculate it with:

$$D_N(R_0, R_1) = \sum_i \max(0, -n_i)$$

It is important to understand why we prefer the L_1 (or Manhattan) distance over the more common L_2 (or Euclidean) distance. It has been shown that the L_1 metric shows better contrast than L_2, especially in high dimensions [Aggarwal et. al. 2001]. As a result, we expect that methods based on the L_1 distance will scale better. It also helps that the L_1 metric requires fewer expensive floating point operations than the L_2 metric and allows for more aggressive culling during searches.

Now, let us describe some implementation details regarding how the insertion heuristic and 2-opt refinement might be efficiently executed using a binary tree.

Let us we implement a binary search tree where each node contains a neighborhood in the tour and the bounding box of the sub-tree rooted at the node. Furthermore, imagine that we are searching the tree in an attempt to conduct an insertion of a neighborhood R_x, as defined for insertion heuristics. We calculate a metric for the right sub-tree, having bound R_{right}:

$$f_N(R_x, R_{right}) = 2\, D_N(R_x, R_{right})$$

We calculate the metric analogously for the left as well. The sub-tree with the lower f_N value is searched first. In the case of ties, the sub-tree whose bound has the smallest furthest distance from R_x is searched first. As we search for an arc to replace, we store the change in tour length for the best arc found so far. If that value is less than the f_N metric for a sub-tree, then that sub-tree does not need to be searched. The f_N calculation serves two purposes, traversal ordering and culling.

Now, imagine that we are searching a similar binary tree for a 2-opt flip. Assume that we have selected an arc, and that we are searching for a partner for that arc. Replacing our selected arc and its partner with a 2-opt flip is intended to shorten the tour. The binary tree nodes have next-node and previous-node pointers, so that each node can easily access the two arcs that connect to its neighborhood. When we reach a node during traversal, we test the incoming arc if the node has a left child; we test the outgoing arc if the node has a right child. This insures that each edge is never tested more than once during a traversal.

If R_{x-} and R_{x+} are the neighborhoods at the beginning and end of our selected arc, then we calculate a metric for the right sub-tree:

$$f_F(R_{x-}, R_{x+}, R_{right}) = D_N(R_{x-}, R_{right}) + D_N(R_{x+}, R_{right}) - D_F(R_{x-}, R_{x+})$$

An f_F value can be calculated for the left sub-tree analogously. The nearest distance values estimate the potential arcs created by a 2-opt flip, while the furthest distance value is a convenient over-estimation of the length of a partner arc inside the bounding volume of the sub-tree. The furthest distance term is constant during traversal, so its value can be cached.

If R_{y-} and R_{y+} are the beginning and end neighborhoods of our best partner found so far, then we store this value:

$$f_B(R_{x-}, R_{x+}, R_{y-}, R_{y+}) = D_F(R_{x-}, R_{y-}) + D_F(R_{x+}, R_{y+}) - D_F(R_{y-}, R_{y+})$$

We search the sub-tree with the lower f_F first. If the f_B value is less than the f_F for a sub-tree, then that sub-tree does not need to be searched. As before, the one value serves two purposes, traversal ordering and culling.

This is an example where the L_1 metric performs best. If L_2 metrics were used, then the f_F calculation that we have defined would not find all possible opportunities for a 2-opt refinement. Here is a more general calculation, which will find any opportunity for a 2-opt refinement when using arbitrary L_k-based metrics (specified for the right sub-tree):

$$f_{Fk} = D_N(R_{x-}, R_{right}) + D_N(R_{x+}, R_{right}) - D_F(R_{right}, R_{right})$$

Of course the nearest distance and furthest distance functions would need to be adjusted to reflect the appropriate L_k distance. With f_{Fk}, the furthest distance term gives the longest possible length of an arc in the bounding box; the length of the box diagonal. Of course, the f_{Fk} calculation does not cull searches as well as f_F when using L_1 metrics.

4 A New TSPN Method

Our technique merges the insertion method and 2-opt refinement with a fast data structure. We strive for sub-linear cost per insertion. We also generalize the interface, so that the solution is suitable for the TSPN, and so that we can delete and insert neighborhoods at will. The tour is coded into a binary search tree, so that an in-order traversal of the tree gives the tour.

Ours is not a complete DTSP method, because we cannot accommodate an arbitrary time-varying distance-cost matrix. However, our method is applicable to geometric problems where a neighborhood moves or changes its shape over time.

To begin, an arc is sought for replacement as with other insertion methods. Our tour is encoded in a binary tree, where each tree node contains a neighborhood and an axis-aligned bounding box for itself and all nodes rooted under it. This allows fast culling of the search using the technique described in the last section.

Before we insert the new neighborhood, we do another search to potentially perform a 2-opt refinement. We seek a partner to the selected arc which is suitable for a 2-opt flip. Again, the binary tree allows us to cull the search as previously described. The search is conducted bottom-up; this is possible because one of the two neighborhoods on every arc is contained in a leaf node. This is a general property of binary search trees. During the search, we can infer the relative rank order between an arc and its partner, so the *between*() function used in some implementations is eliminated [Fredman et al. 1995]. If we find a suitable partner arc that will allow us to decrease tour length, then we perform the flip. This shortens the tour, but it also eliminates the arc that we needed for insertion; so if the flip occurs, then we repeat the process from the beginning. When we finally find an arc that would not benefit from a 2-opt flip, we replace it with new arcs to incorporate the new neighborhood.

To delete a neighborhood, the two arcs adjacent to it are replaced with one arc that reconnects the tour. Then we do a search to find a 2-opt partner for this new arc, applying a flip to refine the tour if a suitable one is found. Deletion shares key routines with insertion, but executes more quickly.

Yan et al. described a DTSP heuristic with a similar structure, interleaving operations from other methods [2004]. Their insertion used primitive operations from genetic-evolutionary methods.

Bentley described a fast 2-opt method using kd-trees to find partner arcs [1990]. His search compares city positions, not arcs, finding nearest-neighbors that might be joined by a flip. His technique is not thorough, in that the culling calculation he uses is too tight; it may result in missed opportunities for 2-opt refinement. Our technique will initiate a 2-opt flip if any such opportunity exists.

The length of tours produced with our method is not worse than (lgN+1) times the optimal [Rosenkrantz et. al. 1977]; and under random insertion, Azar showed a worst-case tour length of Ω(lglgN/lglglgN) times the optimal [1994]. Our method is bound by these results, because it is essentially an insertion heuristic. The 2-opt refinement improves quality, but does not raise it to the levels of a more complete 2-opt solution, which continues flipping arcs throughout the tour until no more flips are possible. Theoretical results for TSP methods often do not reflect real-world performance so our tests are described in the next section.

The nodes in our tree maintain links to adjacent neighborhoods in the tour, so performing flips in constant time with flip-bits stored in nodes, is not straight-forward [Fredman et al. 1995]. As a result, a flip can take linear time in our implementation. The run-time of operations could be expensive because each flip is expensive, and multiple flips could be required for each insertion. We will demonstrate through tests in the next section that each operation appears to run in sub-linear amortized time in practice.

Each time a rotation occurs in the binary tree, node bounds need to be recomputed. As a result, a rotation is an expensive operation. The balancing method of shuffle trees has proven to be effective at improving the scalability of our method. Spatial searches back-track their traversals in a way that ordinary associative data structure searches do not. The first time the depth exceeds the counter, the rotation pivot is recorded. Because depth-first traversal is aggressively ordered with proximity metrics, this rotation pivot should appear along a path to a reasonable solution to the query. Experimental results relating to our balancing method are given in the next section.

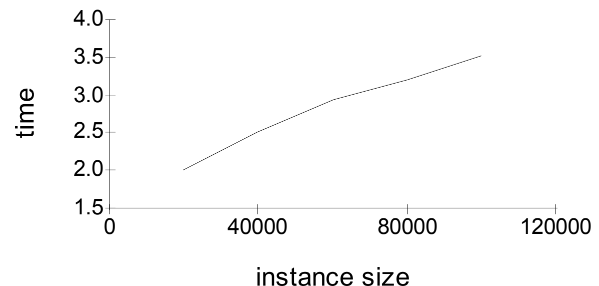
A reference implementation of our dynamic TSPN method is available at the author's website at http://website/website.com.

5 Results

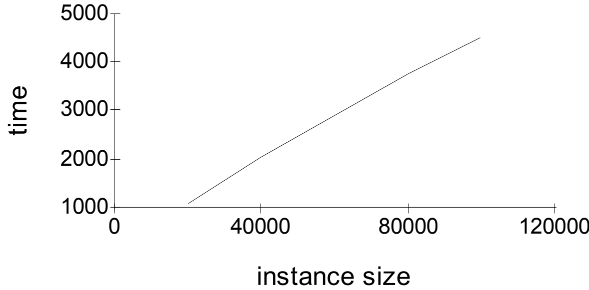
It is common for theoretical bounds for TSP techniques to be poorly correlated to the real-world performance [Johnson and McGeoch 2002]. In this section we describe empirical tests we conducted and present their results.

In our implementation we relaxed the constraint that the tour begins and ends at the same position, but this would be trivial to enforce. Recall also that our distance metrics are based on L_1 distance instead of L_2. When we present tour lengths, the lengths are calculated as the sum of the furthest distances between the two neighborhoods of each arc.

First, we plot a normalized time-cost of an insertion operation on a random 2D neighborhood.



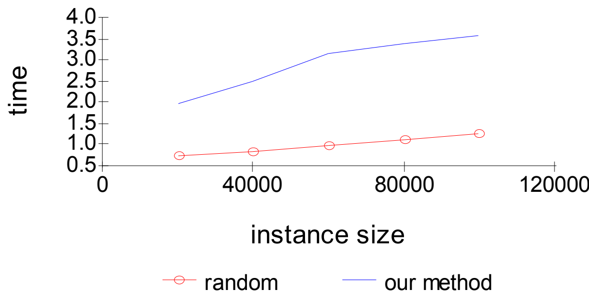
The time-cost per insertion appears to be sub-linear. This demonstrates that our method scales well with data inserted randomly. The next graph shows random insertion of neighborhoods in 16-dimensional space.



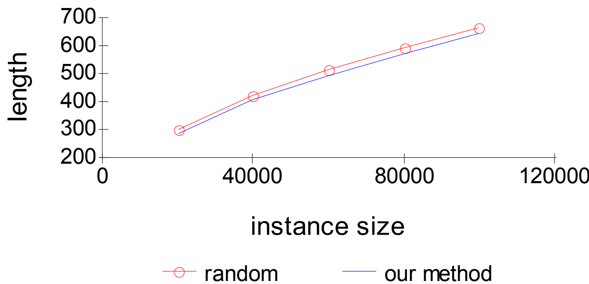
In 16 dimensions, the cost of an insertion scales almost linearly. There is some deterioration in the scalability of our technique as dimension of the space increases, but the method is still suitable for a wide range of applications.

Next, we compare the performance of our method against that of random insertion. Neighborhoods with randomly distributed values are entered into the two tours in the same randomized order. Our implementation of the random insertion heuristic is based on the same code as our new method. It shares the fast search culling technique from section 3. The differences are that no 2-opt refinement occurs and no shuffle tree balancing is used. It is a property of binary trees that randomly inserted elements tend to probabilistically balance the tree, so the tree should be somewhat balanced by the insertion pattern.

First we present the normalized speed of an insertion in 2D space:



Below, we graph the tour lengths produced by each method:



It is not surprising that random insertion performs faster, because it does no 2-opt searches or flips. However, the graph suggests that our method may scale better to very large problem instances. Also, our method consistently produced tours with lengths between 3% and 6% better than random insertion. In the following figure, we visualize results on a relatively small problem instance.

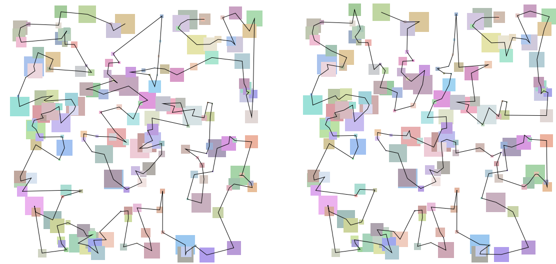
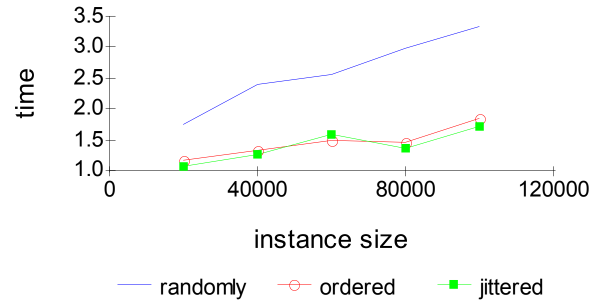


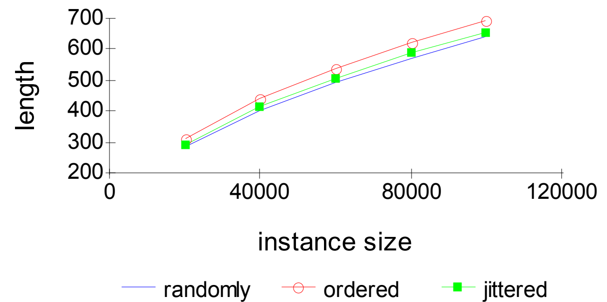
Figure 4: The tour on the left is built with the random insertion heuristic. The tour on the right is built with our method. While our tour is clearly not optimal, its quality is significantly better.

We observed some interesting results when neighborhoods were inserted in non-random order. We began the test by inserting neighborhoods randomly into a tour. Once complete, we inserted the neighborhoods into a second tour in the rank order that they appear in the first. For a third test, we jittered neighborhood rank within +/- 25 versus the order that they appear in the first tour. This jittered rank ordering was used to give the order of insertion for the third tour.

Below we plot the normalized speed of an insertion in 2D space:



Below, we plot the tour lengths produced by each method:



Having some order in the insertion pattern helped improve runtime, but it hurt the quality. The ordered tour was almost 9% longer than the randomly-ordered tour. However, even modest levels of entropy were sufficient to produce good-quality tours. The jittered-order tour was within 2-3% of the length of the randomly-ordered tour and it often ran faster than the others. Our technique is intended to allow for dynamic insertion and deletion

of neighborhoods; however, the quality of the technique clearly benefits from some entropy in the access pattern.

We conducted a test to compare the performance of our method based on shuffle trees against versions implemented with splay trees and unbalanced trees. Since the test used random insertion order, the unbalanced tree was still probabilistically balanced. In the splaying data structure, the deepest node before the first back-track in the search was identified and that node was splayed.

The following table shows the normalized time for an insertion on a problem instance of 100k neighborhoods in varying dimensions:

dim	shuffle	splay	unbalanced
2	3.59	3.51	3.72
4	58.54	59.22	59.78
8	718.09	756.63	729.22
16	4,507.63	4,735.13	4,563.88

In two dimensions, shuffling was not faster than splaying. However, in higher dimensions, shuffling outperformed splaying because the cost of a rotation increased. During each rotation, node bounds need to be recalculated; and in high dimensions, this makes a rotation expensive. The unbalanced tree never outperformed the shuffle tree. Balancing is not the bottleneck of the method, so the differences were significant but not extreme.

Recall that each node in the binary tree has a box which gives bounds for the entire sub-tree rooted at the node. This makes it trivial to perform general spatial queries using the tree. Effectively, the TSPN clusters data for the spatial search. In our testing, the tree performed well for nearest-neighbor searches, where ordering traversal helps to improve run-time. However, for region-intersection tests, which don't benefit from traversal ordering, the tree did not outperform existing methods. The insertion cost and the large overlap between bounding boxes near the root of the tree impaired its performance as a general search structure. It is interesting to note, however, that short sub-sections of the tour seemed to demonstrate bounds that were narrow in many dimensions and wide in only a few. We attribute this to the use of metrics based on the L_1 distance. This shape resembles those found in data structures designed for high-dimensional search [e.g. Berchtold 1998]. Whether or not the overlap between sub-tree bounds near the root can be controlled, adapting the tree for general high-dimensional spatial search, is a topic for future research.

Our TSPN method runs slower than the random insertion heuristic, but consistently showed better tour quality under equal conditions. We observed sub-linear performance for each operation, so we recommend our method as a good compromise between speed and quality. Also, our method is flexible, applicable to dynamic geometric problem instances. Our technique is easy to understand and implement.

6 Conclusions

We have introduced a new method of approximating solutions to the Traveling Salesman Problem with Neighborhoods, when neighborhoods can be appropriately approximated with axis-aligned bounding boxes. The method also allows neighborhoods to be deleted or inserted into the tour at will. We have demonstrated through testing that our solution performs respectably against other methods in both time and solution quality.

References

AGGARWAL, C., HINNEBURG, A., AND KIEM, D. A. 2001. On the Surprising Behavior of Distance Metrics in High Dimensional Spaces. *Proc. 8th International Conference on Database Theory.* 420-434.

APPLEGATE, D., BIXBY, R., CHVATAL, V., AND COOK, W. 1999. *Finding Tours in the TSP.* Report 99885, Research Institute for Discrete Mathematics, Universitat Bonn.

AZAR, Y. 1994. Lower Bounds for Insertion Methods for TSP. *Combinatorics, Probability and Computing, 3*, 285-292.

BENTLEY, J. L. 1990. Experiments on Traveling Salesman Heuristics. *Proceedings of the first annual ACM-SIAM Symposium on Discrete Algorithms*. 91-99.

BERCHTOLD, S., BOHM, C., AND KRIEGEL, H. 1998. The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. *ACM SIGMOD Record, 27*, 2, 142-153.

CHANDRA B., KARLOFF, H., AND TOVEY, C. 1994. New Results on the Old k-Opt Algorithm for the TSP. *Proceedings of the fifth annual ACM-SIAM Symposium on Discrete Algorithms*. 150-159.

ENGLERT, M., ROGLIN, H., AND VOCKING, B. 2007. Worst Case and Probabilistic Analysis of the 2-Opt Algorithm for the TSP. *Proceedings of the eighteenth annual ACM-SIAM Symposium on Discrete Algorithms*. 1295-1304.

FREDMAN, M.L., JOHNSON, D. S., MCGEOCH, L. A., AND OSTHEIMER, G. 1995. Data Structures for the Traveling Salesman. *Journal of Algorithms*, *18*, 432-479.

GALPERIN, I. AND RIVEST, R. 1993. Scapegoat Trees. *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete Algorithms*. 165-174.

JOHNSON, D. AND MCGEOCH, L. 2002. Experimental Analysis of Heuristics for the STSP. *The Traveling Salesman Problem and its Variants.* Gutin and Punnen (eds). Kluwer Academic Publishers, 369-443.

LIN, S., AND KERNIGHAN, B.W. 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, *21*, 498-516.

PATEL, M. 2007. Similarity Metrics for Bounding Volumes. *ACM SIGGRAPH Posters Session*.

ROSENKRANTZ, D. , STEARNS, R. E., AND LEWIS II, P.M. 1977. An Analysis of Several Heuristics for the Traveling Salesman Problem. *SIAM Journal on Computing, 6*, 3, 563-581.

SLEATOR, D. 1992. An Implementation of Top-Down Splaying. ftp://ftp.cs.cmu.edu/user/sleator/splaying/.

SLEATOR, D. AND TARJAN, R. 1985. Self-Adjusting Binary Search Trees. *Journal of the ACM, 32*, 3, 652-686.

YAN, X., KANG, L., CAI, Z., AND LI, H. 2004. An Approach to Dynamic Traveling Salesman Problem. *Proceedings of the third International Conference on Machine Learning and Cybernetics*. 26-29.